Contents

[Intro 1](#_Toc477098822)

[Tools Used 1](#_Toc477098823)

[Detection Methods 1](#_Toc477098824)

[Haar Cascade Detection 2](#_Toc477098825)

[Blob/Background Subtraction 2](#_Toc477098826)

[What I Chose 2](#_Toc477098827)

[Tracking Methods 3](#_Toc477098828)

[OpenCV Algorithms 3](#_Toc477098829)

[My Decision 3](#_Toc477098830)

[Assigning ID’s 3](#_Toc477098831)

[Results 4](#_Toc477098832)

[Conclusion 5](#_Toc477098833)

# Intro

Within the realm of visual computing “detection” and “tracking”, many methodologies and techniques exist to try and solve the problem of allowing a computer to detect, and subsequently follow (or track) objects as they move throughout the field of view, or the frame of a video recording. In this project report, I explain various methods for solving the task of detecting and tracking vehicles and pedestrians, as well as explain how I implemented a system for performing this task on input video streams.

All code/documents/videos related to this project can be found on GitHub at: <https://github.com/rmellmer/Pedestrian-Vehicle-Tracking>

# Tools Used

In creating my detector/tracker, I utilized Python 3.6 and the OpenCV 3.10 library.

# Detection Methods

There are a few common methods to detect prominent and moving objects from within a video. OpenCV assists by allowing two methods natively, namely blob/background detection tracking, and haar cascade detection. Both methods have pros and cons, and these are described in detail below.

## Haar Cascade Detection

Haar cascade detection works by using a pre-trained classifier in the form of a XML file of detection values and metrics, and is loaded into OpenCV for use in determining what type of objects to check for. This method iterates through the entire video frame using a pre-defined window size, and scans for objects that match the Haar classifier description. The results of this method are rectangle bounding boxes surrounding all the detected objects within the frame. This method is great for accurately detecting specific types of objects, and for distinguishing between them. For example, this method could tell a pedestrian from a vehicle/car. One downside to this method is that detection can be quite slow, as the entire frame needs to be iterated over multiple times to tell if the classifier matches any objects within the frame. This makes runtime performance pretty laggy, especially when running on a system without a dedicated graphics processor (like my laptop system I was developing this project on).

Source: http://memememememememe.me/assets/posts/training-haar-cascades/haarFace.jpg

## Blob/Background Subtraction

Background subtraction (also called “Blob” detection) is another common method for detecting prominent objects in a video stream. The basic idea behind this method is to remove “static” pixels from a video stream (e.g. an unchanging background) based on the amount of change that occurs frame-to-frame of a video. This results in leaving moving objects in the video, and OpenCV returns them in the form of “contours”, which is a collection of points that define an outline around the detected object. This method is great for runtime performance on limited hardware, as it doesn’t require complex lookups or detections like the Haar cascade method does. It also allows detection of all moving objects from within a video stream, not just the ones that were specified in the Haar classifier. However, this is also a disadvantage, as background subtraction can’t differentiate between object types like the Haar detection method can.

## What I Chose

For my implementation, I decided to use the background subtraction method. For my purposes, I was determined to find a solution that was runtime efficient, and could detect moving objects in real-time. My development system doesn’t contain a dedicated GPU, so my graphics performance is rather limited, making Haar cascade detection very slow on my machine, which was another reason for my choice of background subtraction to detection objects within my input videos.

# Tracking Methods

## OpenCV Algorithms

OpenCV supplies a robust tracking object API that exposes 5 different tracking algorithms for use. These algorithms are:

**MIL (Multiple Instance Learning)** – Slow, but accurate

**BOOSTING** – A decade old. Mediocre compared to others.

**MEDIANFLOW** – Very fast, but somewhat inaccurate in my experience.

**TLD (Tracking, Learning, and Detection)** – I experienced many detections that shouldn’t have occurred.

**KCF (Kernelized Correlation Filters)** – Good balance of accuracy and speed.

## My Decision

I ran a lot of tests with all five different tracking algorithms listed above, and in my personal experience, KCF performed the best for my applications. KCF seemed to be a good balance between fast performance, which was one of my personal goals for this project, as well as good quality tracking of ROI (region of interest) points.

## Assigning ID’s

Assigning ID’s proved to be a bit more challenging than I was initially expecting it would be. In the end, I eventually developed my own algorithm for assigning ID’s to objects. The pseudocode for my object tracking and identification from frame-to-frame occurs as follows:

First, detect all objects in current frame

For each object:

If a detected object from the previous frame is found within an arbitrary threshold distance, then assume this object is the same object, and assign its ID to us

If not, then this is a new object, and assign it a unique ID

Keep a list of all objects currently being tracked. If an object has been inactive for > 100 frames, delete it from the list

This effectively keeps track of all objects frame-to-frame, and checks if an object in the current frame is within a pre-set threshold distance from an object in the previous frame’s object list. If so, we will assume this object is the same as the previous object. If not, we assume it is a new object, and we initialize the object with a new set of tracking points, and a new, unique ID number.

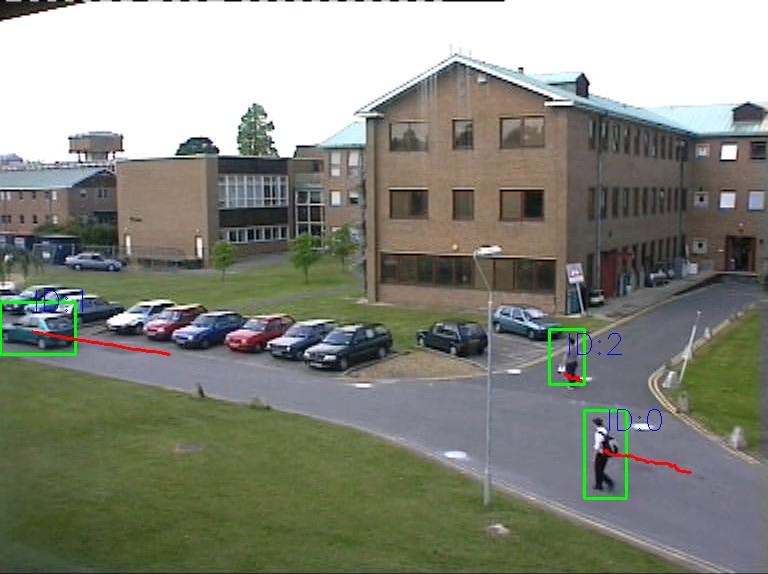
# Results

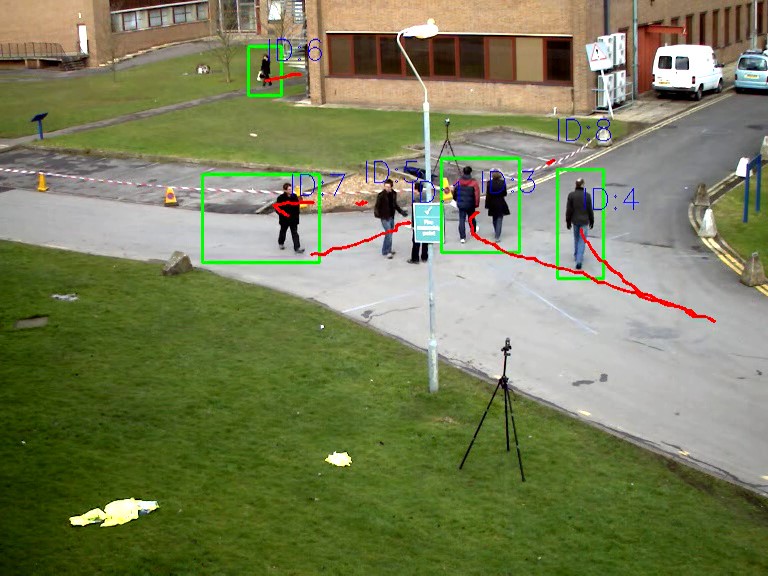
A demo video of my script and some result videos can be found at <https://www.youtube.com/watch?v=_VhF1jRkRTI>.

NOTE: First section of the demo video lags because screen recording software caused video analysis to lag.

Each tracked object also keeps a list of tracked center points, for drawing trails of the object as it moves through the video. Screenshots of result videos can be seen below:







# Conclusion

I tried many different approaches to solving the problem of being able to detect and track moving pedestrians and moving vehicles before I finalized the implementation that I have today. Many nights were spent experimenting with different algorithms and methods, but the combination of background subtraction for object detection, KCF for object tracking, and my own frame-to-frame algorithm for assigning and re-associating unique ID’s fulfilled my requirements for a runtime efficient solution that was accurate, and could detect and track in real-time as the video was being processed. I have learned a ton about the theory and application behind video object analysis and tracking, and am quite happy with the results I achieved.